

11

Configuring an ASP.NET Core application

This chapter covers

- Loading settings from multiple configuration providers
- Storing sensitive settings safely
- Using strongly typed settings objects
- Using different settings in different hosting environments

In part 1 of this book, you learned the basics of getting an ASP.NET Core app up and running and how to use the MVC design pattern to create a traditional web app or a Web API. Once you start building real applications, you will quickly find that you want to tweak various settings at deploy time, without necessarily having to recompile your application. This chapter looks at how you can achieve this in ASP.NET Core using configuration.

I know. Configuration sounds boring, right? But I have to confess, the configuration model is one of my favorite parts of ASP.NET Core. It's so easy to use and so much more elegant than the previous version of ASP.NET. In section 11.3 you'll learn how to load values from a plethora of sources—JSON files, environment variables, and command-line arguments—and combine them into a unified configuration object.

On top of that, ASP.NET Core brings the ability to easily bind this configuration to strongly typed *options* objects. These are simple POCO classes that are populated from the configuration object, which you can inject into your services, as you'll see in section 11.4. This lets you nicely encapsulate settings for different features in your app.

In the final section of this chapter, you'll learn about the ASP.NET Core *hosting environments*. You often want your app to run differently in different situations, such as when running on your developer machine compared to when you deploy it to a production server. These different situations are known as *environments*. By letting the app know in which environment it's running, it can load a different configuration and vary its behavior accordingly.

Before we get to that, let's go back to basics: what is configuration, why do we need it, and how does ASP.NET Core handle these requirements?

11.1 Introducing the ASP.NET Core configuration model

In this section I'll provide a brief description of what we mean by configuration and what you can use it for in ASP.NET Core applications. Configuration is the set of external parameters provided to an application that controls the application's behavior in some way. It typically consists of a mixture of *settings* and *secrets* that the application will load at runtime.

DEFINITION A *setting* is any value that changes the behavior of your application. A *secret* is a special type of setting that contains sensitive data, such as a password, an API key for a third-party service, or a connection string.

The obvious question before we get started is to consider why you need app configuration, and what sort of things you need to configure. You should normally move anything that you can consider a setting or a secret out of your application code. That way, you can easily change these values at deploy time, without having to recompile your application.

You might, for example, have an application that shows the locations of your brick-and-mortar stores. You could have a setting for the connection string to the database in which you store the details of the stores, but also settings such as the default location to display on a map, the default zoom level to use, and the API key for accessing the Google Maps API, as shown in figure 11.1. Storing these settings and secrets outside of your compiled code is good practice, as it makes it easy to tweak them without having to recompile your code.

There's also a security aspect to this; you don't want to hardcode secret values like API keys or passwords into your code, where they could be committed to source control and made publicly available. Even values embedded in your compiled application can be extracted, so it's best to externalize them whenever possible.

Virtually every web framework provides a mechanism for loading configuration, and the previous version of ASP.NET was no different. It used the `<appsettings>` element in a `web.config` file to store key-value configuration pairs. At runtime you'd use

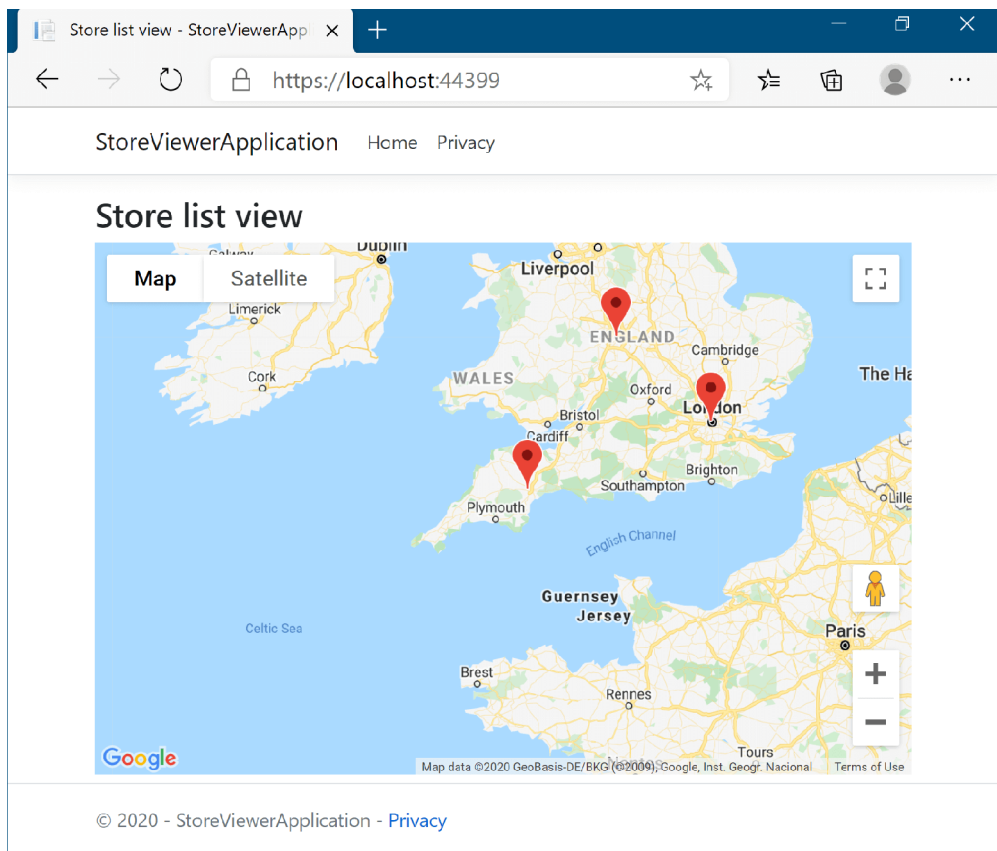


Figure 11.1 You can store the default map location, zoom level, and mapping API Key in configuration and load them at runtime. It's important to keep secrets like API keys in configuration and out of your code.

the static (*wince*) `ConfigurationManager` to load the value for a given key from the file. You could do more advanced things using custom configuration sections, but this was painful and so was rarely used, in my experience.

ASP.NET Core gives you a totally revamped experience. At the most basic level, you're still specifying key-value pairs as strings, but instead of getting those values from a single file, you can now load them from multiple sources. You can load values from files, but they can now be any format you like: JSON, XML, YAML, and so on. On top of that, you can load values from environment variables, from command-line arguments, from a database, or from a remote service. Or you could create your own custom *configuration provider*.

DEFINITION ASP.NET Core uses *configuration providers* to load key-value pairs from a variety of sources. Applications can use many different configuration providers.

The ASP.NET Core configuration model also has the concept of *overriding* settings. Each configuration provider can define its own settings, or it can overwrite settings from a previous provider. You'll see this incredibly useful feature in action in section 11.3.

ASP.NET Core makes it simple to bind these key-value pairs, which are defined as strings, to POCO-setting classes you define in your code. This model of strongly typed configuration makes it easy to logically group settings around a given feature and lends itself well to unit testing.

Before we get into the details of loading configuration from providers, we'll take a step back and look at *where* this process happens—inside `HostBuilder`. For ASP.NET Core 5.0 apps built using the default templates, that's invariably inside the `Host.CreateDefaultBuilder()` method in `Program.cs`.

11.2 Configuring your application with `CreateDefaultBuilder`

As you saw in chapter 2, the default templates in ASP.NET Core 5.0 use the `CreateDefaultBuilder` method. This is an opinionated helper method that sets up a number of defaults for your app. In this section we'll look inside this method to see all the things it configures, and what they're used for.

Listing 11.1 Using `CreateDefaultBuilder` to set up configuration

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The entry point for your application creates an `IHostBuilder`, builds an `IHost`, and calls `Run`.

`CreateDefaultBuilder` sets up a number of defaults, including configuration.

In chapter 2 I glossed over this method, as you will only rarely need to change it for simple apps. But as your application grows, and if you want to change how configuration is loaded for your application, you may find you need to break it apart.

This listing shows an overview of the `CreateDefaultBuilder` method, so you can see how `HostBuilder` is constructed.

Listing 11.2 The `Host.CreateDefaultBuilder` method

```
public static IHostBuilder CreateDefaultBuilder(string[] args)
{
    var builder = new HostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory());
}
```

Creating an instance of `HostBuilder`

The content root defines the directory where configuration files can be found.

Configures application settings, the topic of this chapter	<pre> .ConfigureHostConfiguration(config => { // Configuration provider setup }) </pre>	Configures hosting settings such as determining the hosting environment
	<pre> .ConfigureAppConfiguration((hostingContext, config) => { // Configuration provider setup }) .ConfigureLogging((hostingContext, logging) => { logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging")); logging.AddConsole(); logging.AddDebug(); }) </pre>	Sets up the logging infrastructure
Configures the DI container, optionally enabling verification settings	<pre> .UseDefaultServiceProvider((context, options) => { var isDevelopment = context.HostingEnvironment .IsDevelopment(); options.ValidateScopes = isDevelopment; options.ValidateOnBuild = isDevelopment; }); </pre>	
	<pre> return builder; } </pre>	Returns HostBuilder for further configuration by calling extra methods before calling Build()

The first method called on `HostBuilder` is `UseContentRoot`. This tells the application in which directory it can find any configuration or view files it will need later. This is typically the folder in which the application is running, hence the call to `GetCurrentDirectory`.

TIP `ContentRoot` is *not* where you store static files that the browser can access directly—that's the *WebRoot*, typically `wwwroot`.

The `ConfigureHostConfiguration()` method is where your application determines which `HostingEnvironment` it's currently running in. The framework looks for environment variables and command-line arguments by default, to determine if it's running in a development or production environment. You'll learn more about hosting environments in section 11.5.

`ConfigureLogging` is where you can specify the logging settings for your application. We'll look at logging in detail in chapter 17; for now, it's enough to know that `CreateDefaultBuilder` sets this up for you.

The last method call in `CreateDefaultBuilder`, `UseDefaultServiceProvider`, configures your app to use the built-in DI container. It also sets the `ValidateScopes` and `ValidateOnBuild` options based on the current `HostingEnvironment`. When running the application in the development environment, the app will automatically check for captured dependencies, which you learned about in chapter 10.

The `ConfigureAppConfiguration()` method is the focus of section 11.3. It's where you load the settings and secrets for your app, whether they're in JSON files,

environment variables, or command-line arguments. In the next section, you'll see how to use this method to load configuration values from various configuration providers using the ASP.NET Core `ConfigurationBuilder`.

11.3 Building a configuration object for your app

In this section we'll get into the meat of the configuration system. You'll learn how to load settings from multiple sources, how they're stored internally in ASP.NET Core, and how settings can override other values to give "layers" of configuration. You'll also learn how to store secrets securely while ensuring they're still available when you run your app.

In section 11.2 you saw how the `CreateDefaultBuilder` method can be used to create an instance of `IHostBuilder`. `IHostBuilder` is responsible for setting up many things about your app, including the configuration system in the `ConfigureAppConfiguration` method. This method is passed an instance of a `ConfigurationBuilder`, which is used to define your app's configuration.

The ASP.NET Core configuration model centers on two main constructs: `ConfigurationBuilder` and `IConfiguration`.

NOTE `ConfigurationBuilder` describes how to construct the final configuration representation for your app, and `IConfiguration` holds the configuration values themselves.

You describe your configuration setup by adding a number of `IConfigurationProviders` to the `ConfigurationBuilder` in `ConfigureAppConfiguration`. These describe how to load the key-value pairs from a particular source; for example, a JSON file or environment variables, as shown in figure 11.2. Calling `Build()` on `ConfigurationBuilder` queries each of these providers for the values they contain to create the `IConfigurationRoot` instance.

NOTE Calling `Build()` creates an `IConfigurationRoot` instance, which implements `IConfiguration`. You will generally work with the `IConfiguration` interface in your code.

ASP.NET Core ships with configuration providers for loading data from common locations:

- JSON files
- XML files
- Environment variables
- Command-line arguments
- INI files

If these don't fit your requirements, you can find a whole host of alternatives on GitHub and NuGet, and it's not difficult to create your own custom provider. For

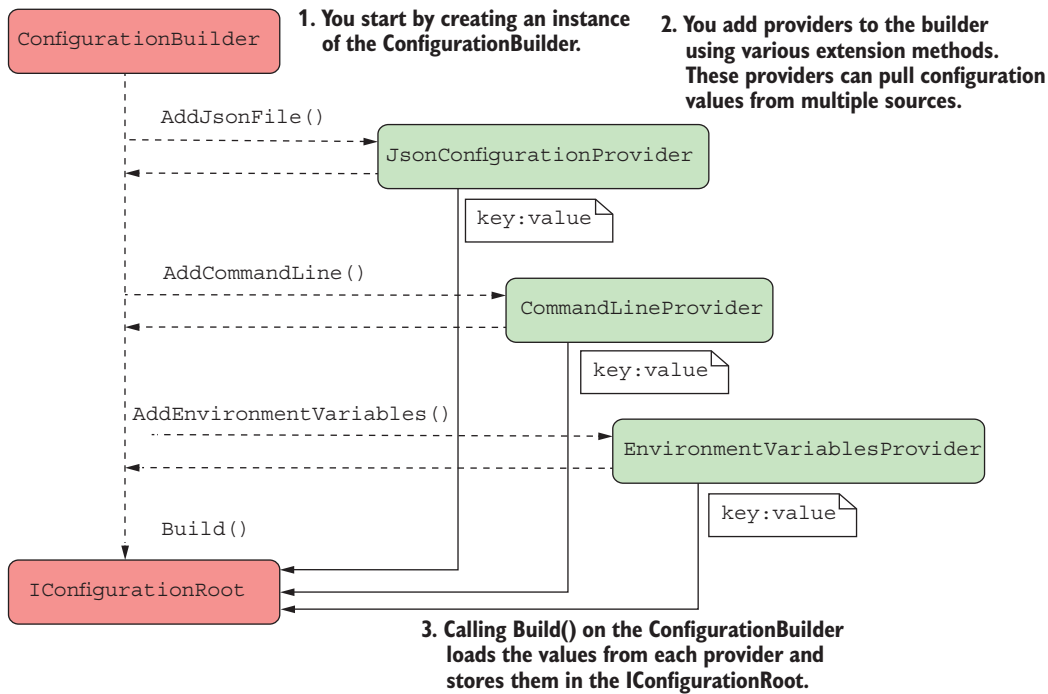


Figure 11.2 Using ConfigurationBuilder to create an instance of IConfiguration. Configuration providers are added to the builder with extension methods. Calling Build() queries each provider to create the IConfigurationRoot, which implements IConfiguration.

example, you could use the official Azure Key Vault provider NuGet package¹ or the YAML file provider I wrote.²

In many cases, the default providers will be sufficient. In particular, most templates start with an appsettings.json file, which contains a variety of settings, depending on the template you choose. The following listing shows the default file generated by the ASP.NET Core 5.0 Web App template without authentication.

Listing 11.3 Default appsettings.json file created by an ASP.NET Core Web template

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}

```

¹ The Azure Key Vault provider is available on NuGet: <http://mng.bz/OEKj>.

² You can find my YAML provider on GitHub at <http://mng.bz/Yqdj>.

```

    },
    "AllowedHosts": "*"
  }
}

```

As you can see, this file mostly contains settings to control logging, but you can add additional configuration for your app here too.

WARNING Don't store sensitive values, such as passwords, API keys, or connection strings, in this file. You'll see how to store these securely in section 11.3.3.

Adding your own configuration values involves adding a key-value pair to the JSON. It's a good idea to "namespace" your settings by creating a base object for related settings, as in the MapSettings object shown here.

Listing 11.4 Adding additional configuration values to an appsettings.json file

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "MapSettings": {
    "DefaultZoomLevel": 9,
    "DefaultLocation": {
      "latitude": 50.500,
      "longitude": -4.000
    }
  }
}

```

Nest all the configuration under the MapSettings key.

Values can be numbers in the JSON file, but they'll be converted to strings when they're read.

You can create deeply nested structures to better organize your configuration values.

I've nested the new configuration inside the MapSettings parent key; this creates a "section" which will be useful later when it comes to binding your values to a POCO object. I also nested the latitude and longitude keys under the DefaultLocation key. You can create any structure of values that you like; the configuration provider will read them just fine. Also, you can store them as any data type—numbers, in this case—but be aware that the provider will read and store them internally as strings.

TIP The configuration keys are *not* case-sensitive in your app, so bear that in mind when loading from providers in which the keys *are* case-sensitive.

Now that you have a configuration file, it's time for your app to load it using ConfigurationBuilder. For this, we'll return to the ConfigureAppConfiguration() method exposed by HostBuilder in Program.cs.

11.3.1 Adding a configuration provider in Program.cs

The default templates in ASP.NET Core use the `CreateDefaultBuilder` helper method to bootstrap `HostBuilder` for your app, as you saw in section 11.2. As part of this configuration, the `CreateDefaultBuilder` method calls `ConfigureAppConfiguration` and sets up a number of default configuration providers, which we'll look at in more detail throughout this chapter:

- *JSON file provider*—Loads settings from an optional JSON file called `appsettings.json`. It also loads settings from an optional environment-specific JSON file called `appsettings.ENVIRONMENT.json`. I'll show how to use environment-specific files in section 11.5.
- *User Secrets*—Loads secrets that are stored safely during development.
- *Environment variables*—Loads environment variables as configuration variables. These are great for storing secrets in production.
- *Command-line arguments*—Uses values passed as arguments when you run your app.

Using the default builder ties you to this default set, but the default builder is optional. If you want to use different configuration providers, you can create your own `HostBuilder` instance instead. If you take this approach, you'll need to set up everything that `CreateHostBuilder` does: logging, hosting configuration, service provider configuration, as well as your app configuration.

An alternative approach is to add *additional* configuration providers by adding an extra call to `ConfigureAppConfiguration`, as shown in the following listing. This allows you to add extra providers on top of those added by `CreateHostBuilder`. In the following listing, you explicitly clear the default providers, which lets you completely customize where configuration is loaded from, without having to replace the defaults `CreateHostBuilder` adds for logging and so on.

Listing 11.5 Loading `appsettings.json` using a custom `HostBuilder`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration(AddAppConfiguration)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });

    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
```

Adds the configuration setup function to `HostBuilder`

`HostBuilder` provides a hosting context and an instance of `ConfigurationBuilder`.

```

{
    config.Sources.Clear();
    config.AddJsonFile("appsettings.json", optional: true);
}

```

← Clears the providers configured by default in `CreateDefaultBuilder`

Adds a JSON configuration provider, providing the filename of the configuration file →

TIP In listing 11.5 I extracted the configuration to a static helper method, `AddAppConfiguration`, but you can also provide this inline as a lambda method.

The `HostBuilder` creates a `ConfigurationBuilder` instance before invoking the `ConfigureAppConfiguration` method. All you need to do is add the configuration providers for your application.

In this example, you've added a single JSON configuration provider by calling the `AddJsonFile` extension method and providing a filename. You've also set the value of `optional` to `true`. This tells the configuration provider to skip over files it can't find at runtime, instead of throwing `FileNotFoundException`. Note that the extension method just registers the provider at this point; it doesn't try to load the file yet.

And that's it! The `HostBuilder` instance takes care of calling `Build()`, which generates `IConfiguration`, which represents your configuration object. This is then registered with the DI container, so you can inject it into your classes. You'd commonly inject this into the constructor of your `Startup` class, so you can use it in the `Configure` and `ConfigureServices` methods:

```

public class Startup
{
    public Startup(IConfiguration config)
    {
        Configuration = config;
    }
    public IConfiguration Configuration { get; }
}

```

NOTE The `ConfigurationBuilder` creates an `IConfigurationRoot` instance, which implements `IConfiguration`. This is registered as an `IConfiguration` in the DI container, *not* an `IConfigurationRoot`. `IConfiguration` is one of the few things that you can inject into the `Startup` constructor.

At this point, at the end of the `Startup` constructor, you have a fully loaded configuration object. But what can you do with it? `IConfiguration` stores configuration as a set of key-value string pairs. You can access any value by its key, using standard dictionary syntax. For example, you could use

```
var zoomLevel = Configuration["MapSettings:DefaultZoomLevel"];
```

to retrieve the configured zoom level for your application. Note that I used a colon (`:`) to designate a separate section. Similarly, to retrieve the latitude key, you could use

```
Configuration["MapSettings:DefaultLocation:Latitude"];
```

NOTE If the requested configuration key does not exist, you will get a null value.

You can also grab a whole section of the configuration using the `GetSection(section)` method, which returns an `IConfigurationSection`, which implements `IConfiguration`. This grabs a chunk of the configuration and resets the namespace. Another way of getting the latitude key would be

```
Configuration.GetSection("MapSettings")["DefaultLocation:Latitude"];
```

Accessing setting values like this is useful in the `ConfigureServices` and `Configure` methods of `Startup`, when you're defining your application. When setting up your application to connect to a database, for example, you'll often load a connection string from the `Configuration` object (you'll see a concrete example of this in the next chapter, when we look at Entity Framework Core).

If you need to access the configuration object like this from classes other than `Startup`, you can use DI to take it as a dependency in your service's constructor. But accessing configuration using string keys like this isn't particularly convenient; you should try to use strongly typed configuration instead, as you'll see in section 11.4.

So far, this is probably all feeling a bit convoluted and run-of-the-mill to load settings from a JSON file, and I'll grant you, it is. Where the ASP.NET Core configuration system shines is when you have multiple providers.

11.3.2 *Using multiple providers to override configuration values*

You've seen that ASP.NET Core uses the Builder pattern to construct the configuration object, but so far you've only configured a single provider. When you add providers, it's important to consider the order in which you add them, as that defines the order in which the configuration values are added to the underlying dictionary. Configuration values from later providers will overwrite values with the same key from earlier providers.

NOTE This bears repeating: the order in which you add configuration providers to `ConfigurationBuilder` is important. Later configuration providers can overwrite the values of earlier providers.

Think of the configuration providers as adding "layers" of configuration values to a stack, where each layer may overlap with some or all of the layers below, as shown in figure 11.3. When you call `Build()`, `ConfigurationBuilder` collapses these layers down into one, to create the final set of configuration values stored in `IConfiguration`.

Update your code to load configuration from three different configuration providers—two JSON providers and an environment variable provider—by adding them to `ConfigurationBuilder`. I've only shown the `AddAppConfiguration` method in listing 11.6 for brevity.

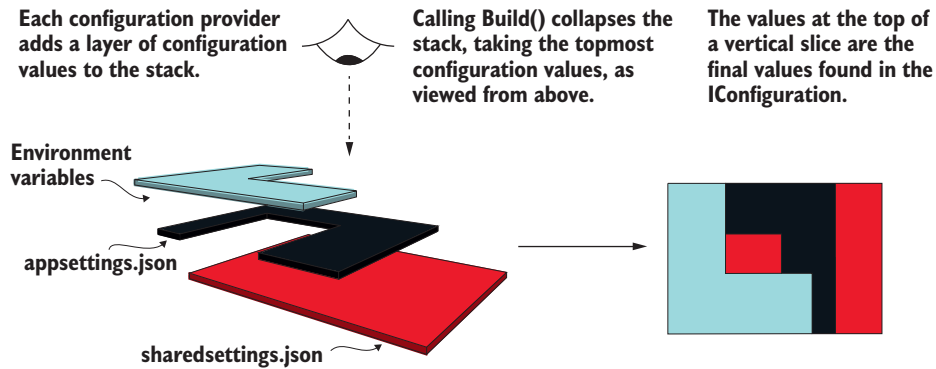


Figure 11.3 Each configuration provider adds a “layer” of values to `ConfigurationBuilder`. Calling `Build()` collapses that configuration. Later providers will overwrite configuration values with the same keys as earlier providers.

Listing 11.6 Loading from multiple providers in `Startup.cs`

```
public class Program
{
    /* Additional Program configuration*/
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.Sources.Clear();
        config
            .AddJsonFile("sharedSettings.json", optional: true)
            .AddJsonFile("appsettings.json", optional: true)
            .AddEnvironmentVariables();
    }
}
```

Loads configuration from a different JSON configuration file before the `appsettings.json` file

Adds the machine's environment variables as a configuration provider

This layered design can be useful for a number of things. Fundamentally, it allows you to aggregate configuration values from a number of different sources into a single, cohesive object. To cement this in place, consider the configuration values in figure 11.4.

Most of the settings in each provider are unique and are added to the final `IConfiguration`. But the `"MyAppConnString"` key appears both in `appsettings.json` and as an environment variable. Because the environment variable provider is added *after* the JSON providers, the environment variable configuration value is used in `IConfiguration`.

The ability to collate configuration from multiple providers is a handy trait on its own, but this design is especially useful when it comes to handling sensitive configuration values, such as connection strings and passwords. The next section shows how

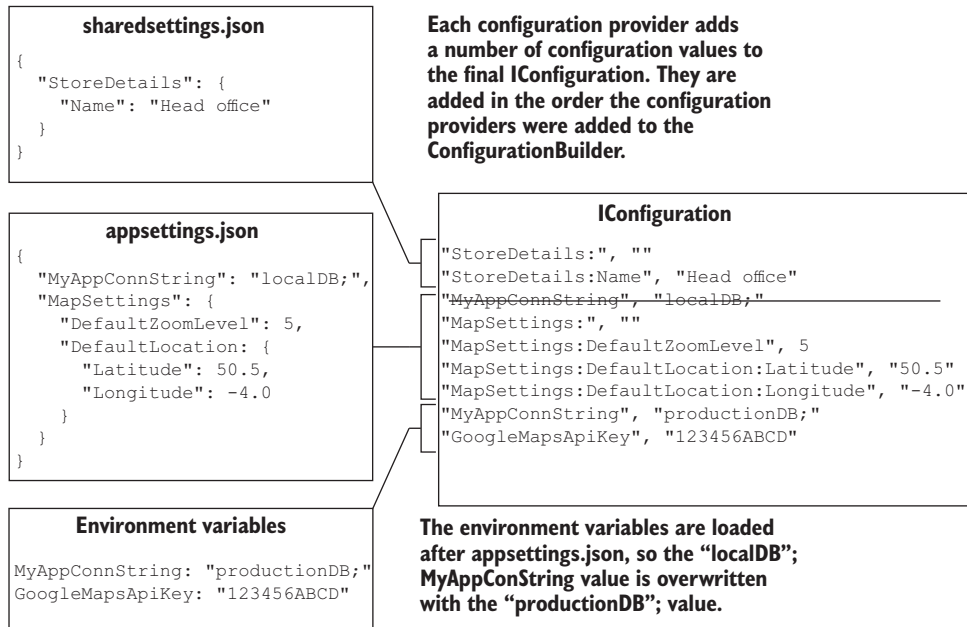


Figure 11.4 The final IConfiguration includes the values from each of the providers. Both appsettings.json and the environment variables include the MyAppConnString key. As the environment variables are added later, that configuration value is used.

to deal with this problem, both locally on your development machine and on production servers.

11.3.3 Storing configuration secrets safely

As soon as you build a nontrivial app, you'll find you have a need to store some sort of sensitive data as a setting somewhere. This could be a password, a connection string, or an API key for a remote service, for example.

Storing these values in appsettings.json is generally a bad idea, as you should never commit secrets to source control; the number of secret API keys people have committed to GitHub is scary! Instead, it's much better to store these values outside of your project folder, where they won't get accidentally committed.

You can do this in a few ways, but the easiest and most commonly used approaches are to use environment variables for secrets on your production server and User Secrets locally.

Neither approach is truly secure, in that they don't store values in an encrypted format. If your machine is compromised, attackers will be able to read the stored values because they're stored in plaintext. They're intended to help you avoid committing secrets to source control.

TIP Azure Key Vault³ is a secure alternative, in that it stores the values encrypted in Azure. But you will still need to use the following approach for storing the Azure Key Value connection details. Another popular option is Vault by Hashicorp (www.vaultproject.io/), which can be run on your premises or in the cloud.

Whichever approach you use to store your application secrets, make sure you aren't storing them in source control, if possible. Even private repositories may not stay private forever, so it's best to err on the side of caution.

STORING SECRETS IN ENVIRONMENT VARIABLES IN PRODUCTION

You can add the environment variable configuration provider using the `AddEnvironmentVariables` extension method, as you've already seen in listing 11.6. This adds all of the environment variables on your machine as key-value pairs in the configuration object.

NOTE The environment variable provider is added by default in `CreateDefaultBuilder` as you saw in section 11.2.

You can create the same hierarchical sections in environment variables that you typically see in JSON files by using a colon (:) or a double underscore (__) to demarcate a section; for example, `MapSettings:MaxNumberOfPoints` or `MapSettings__MaxNumberOfPoints`.

TIP Some environments, such as Linux, don't allow the colon in environment variables. You must use the double underscore approach in these environments instead. The double underscore in environment variables will be converted into a colon when they're imported into the `IConfiguration` object. You should always use the colon when *retrieving* values from an `IConfiguration` in your app.

The environment variable approach is particularly useful when you're publishing your app to a self-contained environment, such as a dedicated server, Azure, or a Docker container. You can set environment variables on your production machine or on your Docker container, and the provider will read them at runtime, overriding the defaults specified in your `appsettings.json` files.⁴

For a development machine, environment variables are less useful, as all your apps would be using the same values. For example, if you set the `ConnectionStrings__DefaultConnection` environment variable, that would be added for *every* app you run locally. That sounds like more of a hassle than a benefit!

³ The Azure Key Vault configuration provider is available as a `Microsoft.Extensions.Configuration.AzureKeyVault` NuGet package. For details on using it in your application, see Microsoft's "Azure Key Vault Configuration Provider in ASP.NET Core" documentation at <http://mng.bz/BR7v>.

⁴ For instructions on how to set environment variables for your operating system, see Microsoft's "Use multiple environments in ASP.NET Core" documentation at <http://mng.bz/d4OD>.

For development scenarios, you can use the User Secrets Manager. This effectively adds per-app environment variables, so you can have different settings for each app but store them in a different location from the app itself.

STORING SECRETS WITH THE USER SECRETS MANAGER IN DEVELOPMENT

The idea behind User Secrets is to simplify storing per-app secrets outside of your app's project tree. This is similar to environment variables, but you use a unique key for each app to keep the secrets segregated.

WARNING The secrets aren't encrypted, so they shouldn't be considered secure. Nevertheless, it's an improvement on storing them in your project folder.

Setting up User Secrets takes a bit more effort than using environment variables, as you need to configure a tool to read and write them, add the User Secrets configuration provider, and define a unique key for your application:

- 1 ASP.NET Core includes the User Secrets provider by default. The .NET SDK also includes a global tool for working with secrets from the command line.
- 2 If you're using Visual Studio, right-click your project and choose **Manage User Secrets**. This opens an editor for a `secrets.json` file in which you can store your key-value pairs, as if it were an `appsettings.json` file, as shown in figure 11.5.

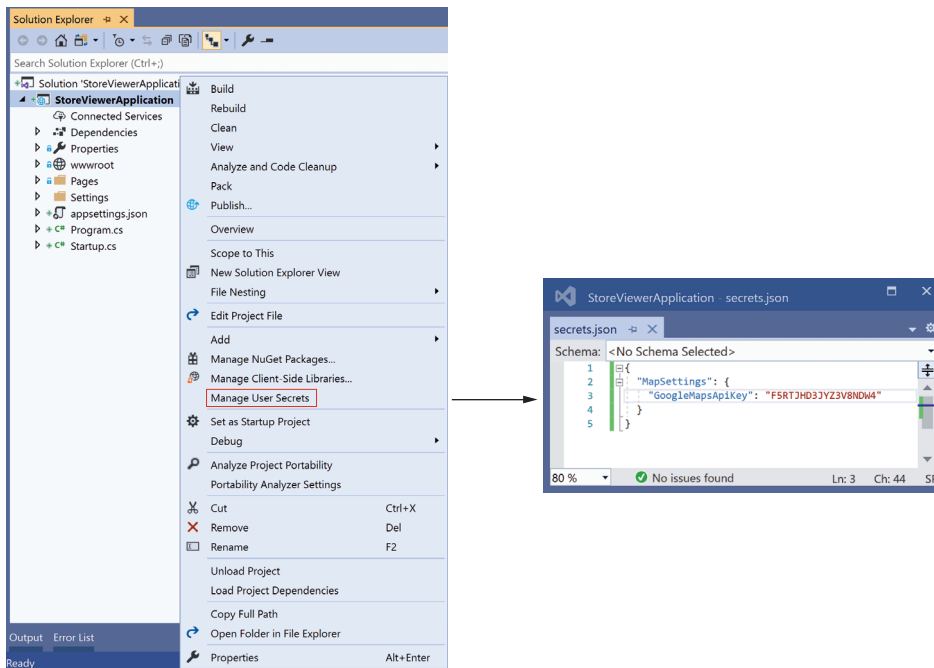


Figure 11.5 Select **Manage User Secrets** to open an editor for the User Secrets app. You can use this file to store secrets when developing your app locally. These are stored outside your project folder, so they won't be committed to source control accidentally.

- 3 Add a unique identifier to your .csproj file. Visual Studio does this automatically when you click Manage User Secrets, but if you're using the command line you'll need to add it yourself. Typically, you'd use a unique ID, like a GUID:

```
<PropertyGroup>
  <UserSecretsId>96eb2a39-1ef9-4d8e-8b20-8e8bd14038aa</UserSecretsId>
</PropertyGroup>
```

- 4 If you aren't using Visual Studio, you can add User Secrets using the command line

```
dotnet user-secrets set "MapSettings:GoogleMapsApiKey" F5RJT9GFHKR7
```

or you can edit the secret.json file directly using your favorite editor. The exact location of this file depends on your operating system and may vary. Check the documentation for details.⁵

Phew, that's a lot of setup, and if you're customizing the `HostBuilder`, you're not done yet! You need to update your app to load the User Secrets at runtime using the `AddUserSecrets` extension method in your `ConfigureAppConfiguration` method:

```
if (env.IsDevelopment())
{
    configBuilder.AddUserSecrets<Startup>();
}
```

NOTE You should only use the User Secrets provider in development, *not* in production, so in the preceding snippet you conditionally add the provider to `ConfigurationBuilder`. In production you should use environment variables or Azure Key Vault, as discussed earlier. This is all configured correctly by default if you use `Host.CreateDefaultBuilder()`.

This method has a number of overloads, but the simplest is a generic method that you can call passing your application's `Startup` class as a generic argument. The User Secrets provider needs to read the `UserSecretsId` property that you (or Visual Studio) added to the .csproj file. The `Startup` class acts as a simple marker to indicate which assembly contains this property.

NOTE If you're interested, the User Secrets package uses the `UserSecretsId` property in your .csproj file to generate an assembly-level `UserSecretsIdAttribute`. The provider then reads this attribute at runtime to determine the `UserSecretsId` of the app, and hence generates the path to the secrets.json file.

⁵ The Secret Manager Tool stores the secrets.json file in the user profile. You can read more about this tool specifically in Microsoft's "Safe storage of app secrets in development in ASP.NET Core" documentation: <http://mng.bz/ryAg>. You can find more about .NET tools in general in Microsoft's "How to manage .NET tools" documentation: <http://mng.bz/VdmX>.

And there you have it—safe storage of your secrets outside your project folder during development. This might seem like overkill, but if you have anything that you consider remotely sensitive that you need to load into configuration, I strongly urge you to use environment variables or User Secrets.

It's almost time to leave configuration providers behind, but before we do, I'd like to show you the ASP.NET Core configuration system's party trick: reloading files on the fly.

11.3.4 *Reloading configuration values when they change*

Besides security, not having to recompile your application every time you want to tweak a value is one of the advantages of using configuration and settings. In the previous version of ASP.NET, changing a setting by editing `web.config` would cause your app to have to restart. This beat having to recompile, but waiting for the app to start up before it could serve requests was a bit of a drag.

In ASP.NET Core you finally get the ability to edit a file and have the configuration of your application automatically update, without having to recompile *or* restart.

An often cited scenario where you might find this useful is when you're trying to debug an app you have in production. You typically configure logging to one of a number of levels:

- Error
- Warning
- Information
- Debug

Each of these settings is more verbose than the last, but it also provides more context. By default, you might configure your app to only log warning and error-level logs in production, so you don't generate too many superfluous log entries. Conversely, if you're trying to debug a problem, you want as much information as possible, so you might want to use the debug log level.

Being able to change configuration at runtime means you can easily switch on extra logs when you encounter an issue and switch them back afterwards by editing your `appsettings.json` file.

NOTE Reloading is generally only available for file-based configuration providers, as opposed to the environment variable or User Secrets provider.

You can enable the reloading of configuration files when you add any of the file-based providers to your `ConfigurationBuilder`. The `Add*File` extension methods include an overload with a `reloadOnChange` parameter. If this is set to `true`, the app will monitor the filesystem for changes to the file and will trigger a complete rebuild of the `IConfiguration`, if needs be. This listing shows how to add configuration reloading to the `appsettings.json` file loaded inside the `AddAppConfiguration` method.

Listing 11.7 Reloading appsettings.json when the file changes

```

public class Program
{
    /* Additional Program configuration*/

    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.AddJsonFile(
            "appsettings.json",
            optional: true
            reloadOnChange: true);
    }
}

```

← **IConfiguration will be rebuilt if the appsettings.json file changes.**

With that in place, any changes you make to the file will be mirrored in the `IConfiguration`. But as I said at the start of this chapter, `IConfiguration` isn't the preferred way to pass settings around in your application. Instead, as you'll see in the next section, you should favor strongly typed POCO objects.

11.4 Using strongly typed settings with the options pattern

In this section you'll learn about strongly typed configuration and the Options pattern. This is the preferred way of accessing configuration in ASP.NET Core. By using strongly typed configuration, you can avoid issues with typos when accessing configuration. It also makes classes easier to test, as you can use simple POCO objects for configuration instead of relying on the `IConfiguration` abstraction.

Most of the examples I've shown so far have been about how to get values *into* `IConfiguration`, as opposed to how to *use* them. You've seen that you can access a key using the `Configuration["key"]` dictionary syntax, but using string keys like this feels messy and prone to typos.

Instead, ASP.NET Core promotes the use of strongly typed settings. These are POCO objects that you define and create and that represent a small collection of settings, scoped to a single feature in your app.

The following listing shows both the settings for your store locator component and display settings to customize the homepage of the app. They're separated into two different objects with "MapSettings" and "AppDisplaySettings" keys, corresponding to the different areas of the app they impact.

Listing 11.8 Separating settings into different objects in appsettings.json

```

{
  "MapSettings": {
    "DefaultZoomLevel": 6,
    "DefaultLocation": {
      "latitude": 50.500,
      "longitude": -4.000
    }
  }
}

```

Settings related to the store locator section of the app

```

    }
  },
  "AppDisplaySettings": {
    "Title": "Acme Store Locator",
    "ShowCopyright": true
  }
}

```

General settings related to displaying the app

The simplest approach to making the home page settings available in the Index.cshtml Razor Page would be to inject IConfiguration into the PageModel and access the values using the dictionary syntax:

```

public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        var title = config["HomePageSettings:Title"];
        var showCopyright = bool.Parse(
            config["HomePageSettings:ShowCopyright"]);
    }
}

```

But you don't want to do this; too many strings for my liking! And that `bool.Parse`? Yuk! Instead, you can use custom strongly typed objects, with all the type safety and IntelliSense goodness that brings.

Listing 11.9 Injecting strongly typed options into a PageModel using `IOptions<T>`

```

public class IndexModel: PageModel
{
    public IndexModel(IOptions<AppDisplaySettings> options)
    {
        AppDisplaySettings settings = options.Value;
        var title = settings.Title;
        bool showCopyright = settings.ShowCopyright;
    }
}

```

The Value property exposes the POCO settings object.

You can inject a strongly typed options class using the `IOptions<>` wrapper interface.

The settings object contains properties that are bound to configuration values at runtime.

The binder can also convert string values directly to primitive types.

The ASP.NET Core configuration system includes a *binder*, which can take a collection of configuration values and *bind* them to a strongly typed object, called an *options class*. This is similar to the concept of model binding from chapter 6, where request values were bound to your POCO binding model classes.

This section shows how to set up the binding of configuration values to a POCO options class and how to make sure it reloads when the underlying configuration values change. We'll also have a look at the different sorts of objects you can bind.

11.4.1 Introducing the *IOptions* interface

ASP.NET Core introduced strongly typed settings as a way of letting configuration code adhere to the single responsibility principle and to allow the injection of configuration classes as explicit dependencies. Such settings also make testing easier; instead of having to create an instance of *IConfiguration* to test a service, you can create an instance of the POCO options class.

For example, the *AppDisplaySettings* class shown in the previous example could be simple, exposing just the values related to the homepage:

```
public class AppDisplaySettings
{
    public string Title { get; set; }
    public bool ShowCopyright { get; set; }
}
```

Your options classes need to be non-abstract and have a public parameterless constructor to be eligible for binding. The binder will set any public properties that match configuration values, as you'll see shortly.

TIP You're not restricted to primitive types like *string* and *bool*; you can use nested complex types too. The options system will bind sections to complex properties. See the associated source code for examples.

To help facilitate the binding of configuration values to your custom POCO options classes, ASP.NET Core introduces the *IOptions<T>* interface. This is a simple interface with a single property, *Value*, which contains your configured POCO options class at runtime. Options classes are set up in the *ConfigureServices* section of *Startup*, as shown here.

Listing 11.10 Configuring the options classes using *Configure<T>* in *Startup.cs*

```
public IConfiguration Configuration { get; }
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MapSettings>(
        Configuration.GetSection("MapSettings"));

    services.Configure<AppDisplaySettings>(
        Configuration.GetSection("AppDisplaySettings"));
}
```

Binds the
MapSettings section
to the POCO options
class *MapSettings*

Binds the *AppDisplaySettings* section to the
POCO options class *AppDisplaySettings*

TIP You don't have to use the same name for both the section and class as I do in listing 11.10; it's just a convention I like to follow. With this convention, you can use the *nameof()* operator to further reduce the chance of typos, such as by calling *GetSection(nameof(MapSettings))*.

Each call to `Configure<T>` sets up the following series of actions internally:

- 1 Creates an instance of `ConfigureOptions<T>`, which indicates that `IOptions<T>` should be configured based on configuration.
If `Configure<T>` is called multiple times, multiple `ConfigureOptions<T>` objects will be used, all of which can be applied to create the final object, in much the same way as `IConfiguration` is built from multiple layers.
- 2 Each `ConfigureOptions<T>` instance binds a section of `IConfiguration` to an instance of the `T` POCO class. This sets any public properties on the options class based on the keys in the provided `ConfigurationSection`.
Remember that the section name ("MapSettings" in listing 11.10) can have any value; it doesn't have to match the name of your options class.
- 3 The `IOptions<T>` interface is registered in the DI container as a singleton, with the final bound POCO object in the `Value` property.

This last step lets you inject your options classes into controllers and services by injecting `IOptions<T>`, as you've seen. This gives you encapsulated, strongly typed access to your configuration values. No more magic strings, woo-hoo!

WARNING If you forget to call `Configure<T>` and inject `IOptions<T>` into your services, you won't see any errors, but the `T` options class won't be bound to anything and will only have default values in its properties.

The binding of the `T` options class to `ConfigurationSection` happens when you first request `IOptions<T>`. The object is registered in the DI container as a singleton, so it's only bound once.

There's one catch with this setup: you can't use the `reloadOnChange` parameter I described in section 11.3.4 to reload your strongly typed options classes when using `IOptions<T>`. `IConfiguration` will still be reloaded if you edit your `appsettings.json` files, but it won't propagate to your options class.

If that seems like a step backwards, or even a deal-breaker, then don't worry. `IOptions<T>` has a cousin, `IOptionsSnapshot<T>`, for such an occasion.

11.4.2 Reloading strongly typed options with `IOptionsSnapshot`

In the previous section, you used `IOptions<T>` to provide strongly typed access to configuration. This provided a nice encapsulation of the settings for a particular service, but with a specific drawback: the options class never changes, even if you modify the underlying configuration file from which it was loaded, such as `appsettings.json`.

This is often not a problem (you shouldn't really be modifying files on live production servers anyway), but if you need this functionality, you can use the `IOptionsSnapshot<T>` interface.

Conceptually, `IOptionsSnapshot<T>` is identical to `IOptions<T>` in that it's a strongly typed representation of a section of configuration. The difference is when, and how often, the POCO options objects are created when each of these are used.

- `IOptions<T>`—The instance is created once, when first needed. It always contains the configuration from when the object instance was first created.
- `IOptionsSnapshot<T>`—A new instance is created, when needed, if the underlying configuration has changed since the last instance was created.

`IOptionsSnapshot<T>` is automatically set up for your options classes at the same time as `IOptions<T>`, so you can use it in your services in exactly the same way. This listing shows how you could update your `IndexModel` home page so that you always get the latest configuration values in your strongly typed `AppDisplaySettings` options class.

Listing 11.11 Injecting reloadable options using `IOptionsSnapshot<T>`

```
public class IndexModel: PageModel
{
    public IndexModel(
        IOptionsSnapshot<AppDisplaySettings> options)
    {
        AppDisplaySettings settings = options.Value;
        var title = settings.Title;
    }
}
```

IOptionsSnapshot<T> will update if the underlying configuration values change.

The Value property exposes the POCO settings object, the same as for IOptions<T>.

The settings object will match the configuration values at some point, instead of at first run.

Whenever you edit the settings file and cause `IConfiguration` to be reloaded, `IOptionsSnapshot<AppDisplaySettings>` will be rebuilt. A new `AppDisplaySettings` object is created with the new configuration values and will be used for all future dependency injection. Until you edit the file again, of course! It's as simple as that; update your code to use `IOptionsSnapshot<T>` instead of `IOptions<T>` wherever you need it.

An important consideration when using the options pattern is the design of your POCO options classes themselves. These are typically simple collections of properties, but there are a few things to bear in mind so that you don't get stuck debugging why the binding seemingly hasn't worked.

11.4.3 Designing your options classes for automatic binding

I've already touched on some of the requirements for POCO classes to work with the `IOptions<T>` binder, but there are a few rules to bear in mind.

The first key point is that the binder will be creating instances of your options classes using reflection, so your POCO options classes need to

- Be non-abstract
- Have a default (public parameterless) constructor

If your classes satisfy these two points, the binder will loop through all the properties on your class and bind any it can. In the broadest sense, the binder can bind any property that

- Is public
- Has a getter—the binder won't write set-only properties

- Has a setter or, for complex types, a non-null value
- Is not an indexer

The following listing shows an extensive options class with a whole host of different types of properties, some of which are valid to bind, and some of which aren't.

Listing 11.12 An options class containing binding and nonbinding properties

<p>The binder will also bind collections, including interfaces; dictionaries must have string keys.</p>	<pre>public class TestOptions { public string String { get; set; } public int Integer { get; set; } public SubClass Object { get; set; } public SubClass ReadOnly { get; } = new SubClass(); public Dictionary<string, SubClass> Dictionary { get; set; } public List<SubClass> List { get; set; } public IDictionary<string, SubClass> IDictionary { get; set; } public IEnumerable<SubClass> IEnumerable { get; set; } public ICollection<SubClass> IEnumerable { get; set; } = new List<SubClass>();</pre>	<p>The binder can bind simple and complex object types, and read-only properties with a default.</p>
<p>The binder can't bind nonpublic, set-only, null-read-only, or indexer properties.</p>	<pre> internal string NotPublic { get; set; } public SubClass SetOnly { set => _setOnly = value; } public SubClass NullReadOnly { get; } = null; public SubClass NullPrivateSetter { get; private set; } = null; public SubClass this[int i] { get => _indexerList[i]; set => _indexerList[i] = value; }</pre>	
<p>These collection properties can't be bound.</p>	<pre> public List<SubClass> NullList { get; } public Dictionary<int, SubClass> IntegerKeys { get; set; } public IEnumerable<SubClass> ReadOnlyEnumerable { get; } = new List<SubClass>(); public SubClass _setOnly = null; private readonly List<SubClass> _indexerList = new List<SubClass>(); public class SubClass { public string Value { get; set; } } }</pre>	<p>The backing fields for implementing SetOnly and Indexer properties—not bound directly</p>

As shown in the listing, the binder generally supports collections—both implementations and interfaces. If the collection property is already initialized, it will use that, but the binder can also create backing fields for them. If your property implements any of the following classes, the binder will create a `List<>` of the appropriate type as the backing object:

- `IReadOnlyList<>`
- `IReadOnlyCollection<>`
- `ICollection<>`
- `IEnumerable<>`

WARNING You can't bind to an `IEnumerable<>` property that has already been initialized, as the underlying type doesn't expose an `Add` function. You *can* bind to an `IEnumerable<>` if you leave its initial value as `null`.

Similarly, the binder will create a `Dictionary<,>` as the backing field for properties with dictionary interfaces, as long as they use string keys:

- `IDictionary<string,>`
- `ReadOnlyDictionary<string,>`

WARNING You can't bind dictionaries with non-string keys, such as `int`. For examples of binding collection types, see the associated source code for this book.

Clearly there are quite a few nuances here, but if you stick to the simple cases from the preceding example, you'll be fine. Be sure to check for typos in your JSON files!

TIP The options pattern is most commonly used to bind POCO classes to configuration, but you can also configure your strongly typed settings classes in code by providing a lambda to the `Configure` function; for example, `services.Configure<TestOptions>(opt => opt.Value=true)`.

The Options pattern is used throughout ASP.NET Core, but not everyone is a fan. In the next section you'll see how to use strongly typed settings and the configuration binder *without* the Options pattern.

11.4.4 Binding strongly typed settings without the *IOptions* interface


The `IOptions` interface is very much canonical in ASP.NET Core—it's used by the core ASP.NET Core libraries and has various convenience functions for binding strongly typed settings, as you've already seen.

In many cases, however, the `IOptions` interface doesn't give many benefits for *consumers* of the strongly typed settings objects. Services must take a dependency on the `IOptions` interface but then immediately extract the “real” object by calling `IOptions<T>.Value`. This can be especially annoying if you're building a reusable library that isn't inherently tied to ASP.NET Core, as you must expose the `IOptions<T>` interface in all your public APIs.

Luckily, the configuration binder that maps `IConfiguration` objects to strongly typed settings objects isn't inherently tied to `IOptions`. Listing 11.13 shows how you can manually bind a strongly typed settings object to a configuration section and register it with the DI container.

Listing 11.13 Configuring strongly typed settings without `IOptions` in `Startup.cs`

```
public IConfiguration Configuration { get; }
public void ConfigureServices(IServiceCollection services)
{
    var settings = new MapSettings ();
```



Create a new instance of
the `MapSettings` object.


```

Configuration.GetSection("MapSettings").Bind(settings);
services.AddSingleton(settings);
}

```

Register the settings object as a singleton.

Bind the MapSettings section in IConfiguration to the settings object.

You can now inject the MapSettings object directly into your services without the additional ceremony required to use `IOptions<MapSettings>`.

```

public class MyMappingController
{
    private readonly MapSettings _settings;
    public MyMappingController(MapSettings settings)
    {
        _settings = settings;
    }
}

```

If you use this approach, you won't benefit from the ability to reload strongly typed settings without further work, or from some of the more advanced usages of `IOptions`, but in most cases that's not a big problem. I'm a fan of this approach generally, but as always, consider what you're losing before adopting it wholeheartedly.

TIP In chapter 19 I show one such scenario, where you configure an `IOptions` object using services in your DI container. For other advanced scenarios see Microsoft's "Options pattern in ASP.NET Core" documentation, <http://mng.bz/DR7y>, or see the various `IOptions` posts on my blog, such as this one: <http://mng.bz/11Aj>.

That brings us to the end of this section on strongly typed settings. In the next section we'll look at how you can dynamically change your settings at runtime, based on the environment in which your app is running.

11.5 *Configuring an application for multiple environments*

In this section you'll learn about hosting environments in ASP.NET Core. You'll learn how to set and determine which environment an application is running in, and how to change which configuration values are used, based on the environment. This lets you easily switch between different sets of configuration values in production compared to development, for example.

Any application that makes it to production will likely have to run in multiple environments. For example, if you're building an application with database access, you'll probably have a small database running on your machine that you use for development. In production, you'll have a completely different database running on a server somewhere else.

Another common requirement is to have different amounts of logging depending on where your app is running. In development it's great to generate lots of logs as it helps debugging, but once you get to production, too much logging can be

overwhelming. You'll want to log warnings and errors, and maybe information-level logs, but definitely not debug-level logs!

To handle these requirements, you need to make sure your app loads different configuration values depending on the environment it's running in: load the production database connection string when in production, and so on. You need to consider three aspects:

- How does your app identify which environment it's running in?
- How do you load different configuration values based on the current environment?
- How can you change the environment for a particular machine?

This section tackles each of these questions in turn, so you can easily tell your development machine apart from your production servers and act accordingly.

11.5.1 Identifying the hosting environment

As you saw in section 11.2, the `ConfigureHostingConfiguration` method on `HostBuilder` is where you define how your application calculates the hosting environment. By default, `CreateDefaultBuilder` uses, perhaps unsurprisingly, an environment variable to identify the current environment. The `HostBuilder` looks for a magic environment variable called `ASPNETCORE_ENVIRONMENT` and uses it to create an `IHostEnvironment` object.

NOTE You can use either the `DOTNET_ENVIRONMENT` or `ASPNETCORE_ENVIRONMENT` environment variables. The `ASPNETCORE_` value overrides the `DOTNET_` value if both are set. I use the `ASPNETCORE_` version throughout this book.

The `IHostEnvironment` interface exposes a number of useful properties about the running context of your app. Some of these you've already seen, such as `ContentRootPath`, which points to the folder containing your application's content files; for example, the `appsettings.json` files. The property you're interested in here is `EnvironmentName`.

The `IHostEnvironment.EnvironmentName` property is set to the value of the `ASPNETCORE_ENVIRONMENT` environment variable, so it can be anything, but you should stick to three commonly used values in most cases:

- "Development"
- "Staging"
- "Production"

ASP.NET Core includes several helper methods for working with these three values, so you'll have an easier time if you stick to them. In particular, whenever you're testing whether your app is running in a particular environment, you should use one of the following extension methods:

- `IHostEnvironment.IsDevelopment()`
- `IHostEnvironment.IsStaging()`

- `IHostEnvironment.IsProduction()`
- `IHostEnvironment.IsEnvironment(string environmentName)`

These methods all make sure they do case-insensitive checks of the environment variable, so you don't get any wonky errors at runtime if you don't capitalize the environment variable value.

TIP Where possible, use the `IHostEnvironment` extension methods over direct string comparison with `EnvironmentValue`, as they provide case-insensitive matching.

`IHostEnvironment` doesn't do anything other than expose the details of your current environment, but you can use it in various ways. In chapter 8 you saw the `Environment Tag Helper`, which you used to show and hide HTML based on the current environment. Now you know where it was getting its information—`IHostEnvironment`.

You can use a similar approach to customize which configuration values you load at runtime by loading different files when running in development versus production. This is common and is included out of the box in most ASP.NET Core templates, as well as in the `CreateDefaultBuilder` helper method.

11.5.2 Loading environment-specific configuration files

The `EnvironmentName` value is determined early in the process of bootstrapping your application, before the `ConfigurationBuilder` passed to `ConfigureAppConfiguration` is created. This means you can dynamically change which configuration providers are added to the builder, and hence which configuration values are loaded when the `IConfiguration` is built.

A common pattern is to have an optional, environment-specific appsettings `.ENVIRONMENT.json` file that's loaded after the default `appsettings.json` file. This listing shows how you could achieve this if you're customizing the `ConfigureAppConfiguration` method in `Program.cs`.

Listing 11.14 Adding environment-specific appsettings.json files

```
public class Program
{
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        var env = hostingContext.HostingEnvironment;
        config
            .AddJsonFile(
                "appsettings.json",
                optional: false)
            .AddJsonFile(
                $"appsettings.{env.EnvironmentName}.json",
                optional: true);
    }
}
```

The current `IHostEnvironment` is available on `HostBuilderContext`.

It's common to make the base `appsettings.json` compulsory.

Adds an optional environment-specific JSON file where the filename varies with the environment

With this pattern, a global `appsettings.json` file contains settings applicable to most environments. Additional, optional JSON files called `appsettings.Development.json`, `appsettings.Staging.json`, and `appsettings.Production.json` are subsequently added to `ConfigurationBuilder`, depending on the current `EnvironmentName`.

Any settings in these files will overwrite values from the global `appsettings.json`, if they have the same key, as you've seen previously. This lets you do things like set the logging to be verbose in the development environment only and switch to more selective logs in production.

Another common pattern is to completely add or remove configuration providers depending on the environment. For example, you might use the User Secrets provider when developing locally, but Azure Key Vault in production. This listing shows how you can use `IHostEnvironment` to conditionally include the User Secrets provider in development only.

Listing 11.15 Conditionally including the User Secrets configuration provider

```
public class Program
{
    /* Additional Program configuration*/
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        var env = hostingContext.HostingEnvironment
        config
            .AddJsonFile(
                "appsettings.json",
                optional: false)
            .AddJsonFile(
                $"appsettings.{env.EnvironmentName}.json",
                optional: true);
        if (env.IsDevelopment())
        {
            builder.AddUserSecrets<Startup>();
        }
    }
}
```

Extension methods make checking the environment simple and explicit.

In Staging and Production, the User Secrets provider won't be used.

It's also common to customize your application's middleware pipeline based on the environment. In chapter 3 you learned about `DeveloperExceptionPageMiddleware` and how you should use it when developing locally. The following listing shows how you can use `IHostEnvironment` to control your pipeline in this way, so that when you're in staging or production, your app uses `ExceptionHandlerMiddleware` instead.

Listing 11.16 Using the hosting environment to customize your middleware pipeline

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }
}
```

```

if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
}

```

In development, `DeveloperExceptionPageMiddleware` is added to the pipeline.

In staging or production, the pipeline uses `ExceptionHandlerMiddleware` instead.

NOTE In listing 11.16 we injected `IWebHostEnvironment` instead of `IHostEnvironment`. This interface extends `IHostEnvironment` by adding the `WebRootPath` property—the path to the `wwwroot` folder in your application. We don’t need that path here, but it’s good to be aware of its existence!

You can inject `IHostEnvironment` anywhere in your app, but I’d advise against using it in your own services, outside of `Startup` and `Program`. It’s far better to use the configuration providers to customize strongly typed settings based on the current hosting environment, and inject these settings into your application instead.

As useful as it is, setting `IHostEnvironment` with an environment variable can be a little cumbersome if you want to switch back and forth between different environments during testing. Personally, I’m always forgetting how to set environment variables on the various operating systems I use. The final skill I’d like to teach you is how to set the hosting environment when you’re developing locally.

11.5.3 *Setting the hosting environment*

In this section I’ll show you a couple of ways to set the hosting environment when you’re developing. This makes it easy to test a specific app’s behavior in different environments without having to change the environment for all the apps on your machine.

If your ASP.NET Core application can’t find an `ASPNETCORE_ENVIRONMENT` environment variable when it starts up, it defaults to a production environment, as shown in figure 11.6. This means that when you deploy to production, you’ll be using the correct environment by default.

TIP By default, the current hosting environment is logged to the console at startup, which can be useful for quickly checking that the environment variable has been picked up correctly.

If the `HostBuilder` can't find the `ASPNETCORE_ENVIRONMENT` variable at runtime, it will default to `Production`.

```

C:\repos\ch11>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\repos\ch11
  
```

Figure 11.6 By default, ASP.NET Core applications run in the production hosting environment. You can override this by setting the `ASPNETCORE_ENVIRONMENT` variable.

Another option is to use a `launchSettings.json` file to control the environment. All the default ASP.NET Core applications include this file in the `Properties` folder. `LaunchSettings.json` defines *profiles* for running your application.

TIP Profiles can be used to run your application with different environment variables. You can also use profiles to emulate running on Windows behind IIS by using the IIS Express profile. Personally, I rarely use this profile, even on Windows, and I always choose the “project” profile.

A typical `launchSettings.json` file is shown in the following listing, which defines two profiles: “IIS Express”, and “StoreViewerApplication”. The latter profile is equivalent to using `dotnet run` to run the project, and it’s conventionally named the same as the project containing the `launchSettings.json` file.

Listing 11.17 A typical `launchSettings.json` file defining two profiles

```

{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:53846",
      "sslPort": 44399
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "StoreViewerApplication": {
      "commandName": "Project",
      "launchBrowser": true,
  
```

Defines settings for when running behind IIS or using the IIS Express profile

If true, launches the browser when you run the application →

The IIS Express profile is used by default in Visual Studio on Windows.

Defines custom environment variables for the profile. Sets the environment to Development.

If true, launches the browser when you run the application →

The “project” profile, equivalent to calling `dotnet run` on the project

```

"environmentVariables": {
  "ASPNETCORE_ENVIRONMENT": "Development"
},
"applicationUrl":
  "https://localhost:5001;http://localhost:5000"
}
}

```

Each profile can have different environment variables.

Defines the URLs the application will listen on in this profile

The advantage of using the `launchSettings.json` file locally is that it allows you to set “local” environment variables for a project. For example, in listing 11.17, the environment is set to the development environment. This lets you use different environment variables for each project, and even for each profile, and store them in source control.

You can choose a profile to use in Visual Studio by selecting from the drop-down list next to the Debug button on the toolbar, as shown in figure 11.7. You can choose a profile to run from the command line using `dotnet run --launch-profile <Profile Name>`. If you don’t specify a profile, the first “project” type profile is used. If you don’t want to use *any* profile, you must explicitly ignore the `launchSettings.json` file by using `dotnet run --no-launch-profile`.

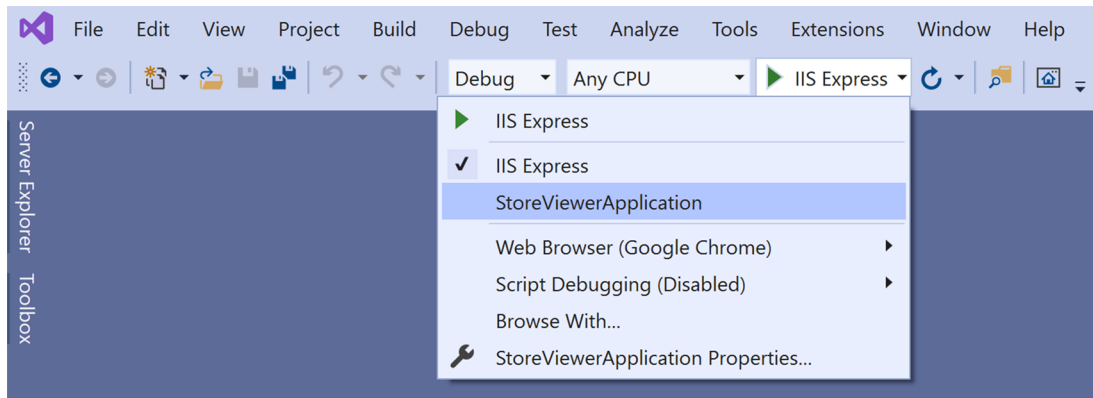


Figure 11.7 You can choose the profile to use from Visual Studio by selecting from the Debug drop-down list. Visual Studio defaults to using the IIS Express profile. The default profile running with `dotnet run` is the first “project” profile—`StoreViewerApplication` in this case.

If you’re using Visual Studio, you can also edit the `launchSettings.json` file visually: double-click the Properties node and choose the Debug tab. You can see in figure 11.8 that the `ASPNETCORE_ENVIRONMENT` is set to development; any changes made in this tab are mirrored in `launchSettings.json`.

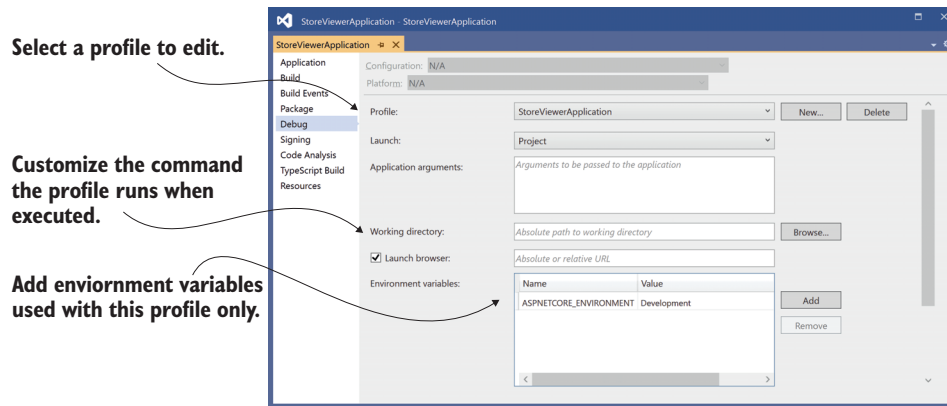


Figure 11.8 You can use Visual Studio to edit the `launchSettings.json` file if you prefer. Changes will be mirrored between the `launchSettings.json` file and the Properties dialog box.

The `launchSettings.json` file is intended for local development only; by default, the file isn't deployed to production servers. While you *can* deploy and use the file in production, it's generally not worth the hassle. Environment variables are a better fit.

One final trick I've used to set the environment in production is to use command-line arguments. For example, you could set the environment to staging like this:

```
dotnet run --no-launch-profile --environment Staging
```

Note that you also have to pass `--no-launch-profile` if there's a `launchSettings.json` file; otherwise the values in the file take precedence.

That brings us to the end of this chapter on configuration. Configuration isn't glamorous, but it's an essential part of all apps. The ASP.NET Core configuration provider model handles a wide range of scenarios, letting you store settings and secrets in a variety of locations.

Simple settings can be stored in `appsettings.json`, where they're easy to tweak and modify during development, and they can be overwritten by using environment-specific JSON files. Meanwhile, your secrets and sensitive settings can be stored outside the project file in the User Secrets manager, or as environment variables. This gives you both flexibility and safety—as long as you don't go writing your secrets to `appsettings.json`!

In the next chapter we'll take a brief look at the new object-relational mapper that fits well with ASP.NET Core: Entity Framework Core. We'll only be getting a taste of it in this book, but you'll learn how to load and save data, build a database from your code, and migrate the database as your code evolves.

Summary

- Anything that could be considered a setting or a secret is normally stored as a configuration value.
- ASP.NET Core uses configuration providers to load key-value pairs from a variety of sources. Applications can use many different configuration providers.
- `ConfigurationBuilder` describes how to construct the final configuration representation for your app, and `IConfiguration` holds the configuration values themselves.
- You create a configuration object by adding configuration providers to an instance of `ConfigurationBuilder` using extension methods such as `AddJsonFile()`. `HostBuilder` creates the `ConfigurationBuilder` instance for you and calls `Build()` to create an instance of `IConfiguration`.
- ASP.NET Core includes built-in providers for JSON files, XML files, environment files, and command-line arguments, among others. NuGet packages exist for many other providers, such as YAML files and Azure Key Vault.
- The order in which you add providers to `ConfigurationBuilder` is important; subsequent providers replace the values of settings defined in earlier providers.
- Configuration keys aren't case-sensitive.
- You can retrieve settings from `IConfiguration` directly using the indexer syntax; for example, `Configuration["MySettings:Value"]`.
- The `CreateDefaultBuilder` method configures JSON, environment variables, command-line arguments, and User Secret providers for you. You can customize the configuration providers used in your app by calling `ConfigureAppConfiguration`.
- In production, store secrets in environment variables. These can be loaded after your file-based settings in the configuration builder.
- On development machines, the User Secrets Manager is a more convenient tool than using environment variables. It stores secrets in your OS user's profile, outside the project folder.
- Be aware that neither environment variables nor the User Secrets Manager tool encrypt secrets, they merely store them in locations that are less likely to be made public, as they're outside your project folder.
- File-based providers, such as the JSON provider, can automatically reload configuration values when the file changes. This allows you to update configuration values in real time, without having to restart your app.
- Use strongly typed POCO options classes to access configuration in your app.
- Use the `Configure<T>()` extension method in `ConfigureServices` to bind your POCO options objects to `ConfigurationSection`.
- You can inject the `IOptions<T>` interface into your services using DI. You can access the strongly typed options object on the `Value` property.

- You can configure `IOptions<T>` objects in code instead of using configuration values by passing a lambda to the `Configure()` method.
- If you want to reload your POCO options objects when your configuration changes, use the `IOptionsSnapshot<T>` interface instead.
- Applications running in different environments, development versus production for example, often require different configuration values.
- ASP.NET Core determines the current hosting environment using the `ASPNETCORE_ENVIRONMENT` environment variable. If this variable isn't set, the environment is assumed to be production.
- You can set the hosting environment locally by using the `launchSettings.json` file. This allows you to scope environment variables to a specific project.
- The current hosting environment is exposed as an `IHostEnvironment` interface. You can check for specific environments using `IsDevelopment()`, `IsStaging()`, and `IsProduction()`.
- You can use the `IHostEnvironment` object to load files specific to the current environment, such as `appsettings.Production.json`.